Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Текстовая часть языка QReal:Robots

курсовая работа студента 371 группы
Клепача Богдана Валентиновича

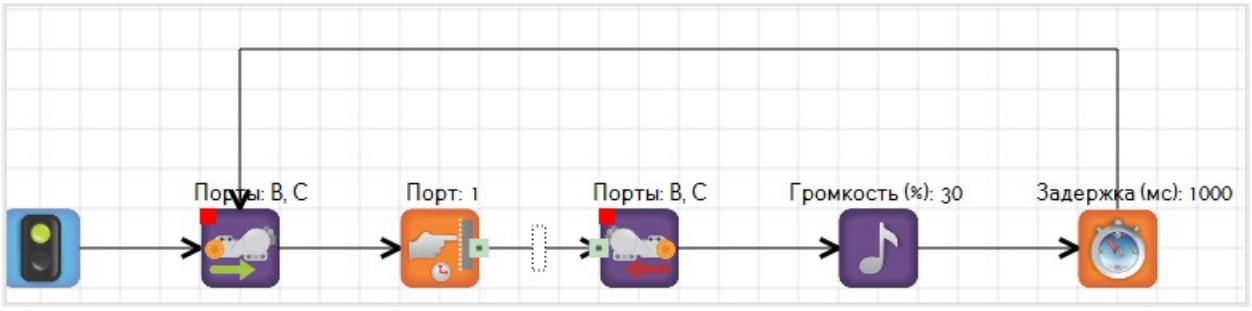
Научный руководитель аспирант кафедры
системного программирования Мордвинов Д.А.

Санкт-Петербург, 2014

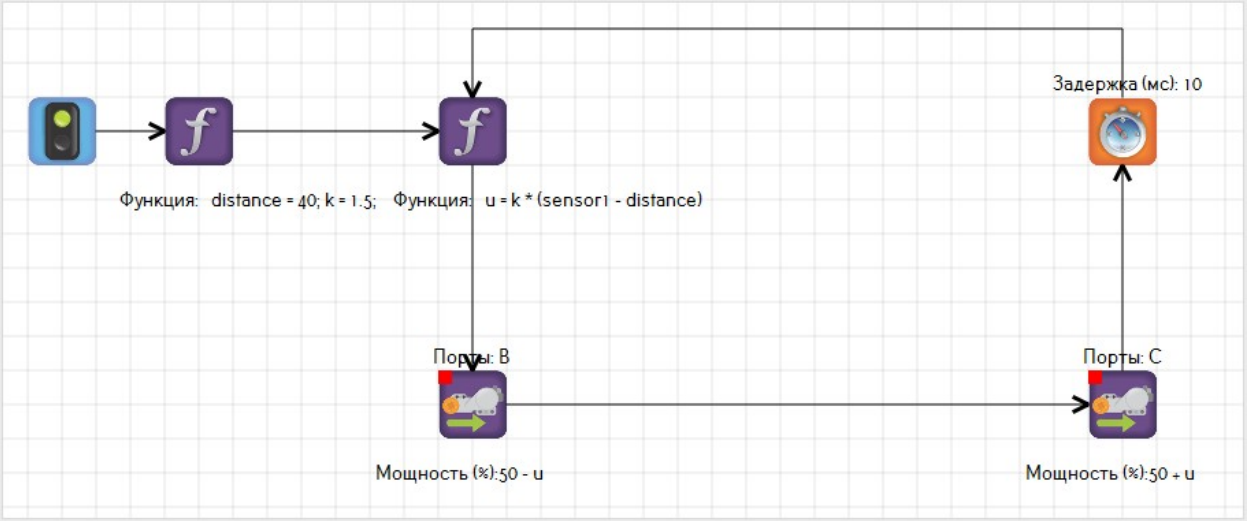
Введение.

QReal:Robots - это среда визуального программирования роботов, предназначенная для обучения программированию. Ученики могут с помощью графических средств создавать программы для различных роботических конструкторов, в том числе Lego NXT и выполнять эти программы прямо на компьютере, либо посылая команды роботу через Bluetooth или по USB, либо при отсутствие робота выполнять программу в 2D модели. Также имеется возможность сгенерировать код для робота.

Программа состоит из набора блоков каждый из которых отвечает за какое-либо действие или проверку условия. Ниже приведен пример программы робота, который в случае столкновения с препятствием, отъезжает назад и подает звуковой сигнал.



Также программа может описываться с помощью функций. В блоке “Функция” можно инициализировать, присваивать и изменять значение переменных. Ниже приведен пример программы для робота, позволяющей ему ехать по линии.



В настоящий момент система типов QReal:Robots поддерживает только целые и вещественные типы чисел. По мере развития проекта появляется необходимость в расширении системы типов. Например, при работе акселерометр выдает набор из трех значений проекций ускорения на различные оси, для работы с ними необходимы массивы. Но текущая реализация синтаксического анализатора и системы вывода типов довольно сырая и сложно модифицируемая.

Постановка задачи.

Необходимо расширить язык, используемый в QReal:Robots. Для этого необходимо реализовать синтаксический анализатор, который выдает дерево разбора и способен разбирать выражения различных типов. Также необходимо научиться использовать дерево разбора для определения типов переменных и генерации кода для различных роботических платформ, которые уже поддерживаются в QReal:Robots.

Для начала требуется более формально определить, требования предъявляемые к языку, синтаксическому анализатору и системе типов. Кроме тех типов, что уже поддерживаются, необходимо поддержать в языке переменные строковых и булевых типов, а также массивы. К примеру, хотелось бы иметь строковые переменные, чтобы выводить информацию на экран. И кроме тех операций, что уже поддерживаются, необходимо поддержать следующие операции:

1. побитовые;
2. взятие элемента по индексу;
3. инкремент и декремент целочисленной переменной.

Необходимо учесть, что данная среда используется для обучения школьников программированию и, следовательно, язык должен быть похож на те языки, которые используются на практике. Для лаконичности языка хотелось бы обойтись без явного приведения типов.

После того как будут реализованы необходимые компоненты, необходимо написать тесты для проверки корректности работы компонент. Тесты должны покрывать все аспекты работы анализатора.

Обзор существующих решений.

Возможны два способа решения поставленной задачи. Первый - это провести исследование среди готовых скриптовых решений и подобрать подходящее нам. Второй способ - это доработка сущесьвующего языка.

Исследование готовых скриптовых решений

Помимо уже описанных требований, необходимо, чтобы для данного языка имелся синтаксический анализатор, интерпретатор и отладчик с открытым кодом, который можно переиспользовать, либо транслятор в один из языков используемых роботическими платформами. Были рассмотрены несколько вариантов :

1. Lua
    a. QtLua
    b. Lua2c
    c. Lua_icxx
    d. ToLua++
2. CSL (C Script Language)

Рассмотрим подробнее каждый из предложенных вариантов.

QtLua

QtLua[7] – это альтернатива QtScript. Она позволяет создавать приложения для Qt4/Qt5 на скриптовом языке Lua. Язык Lua отличается более мощными и гораздо более гибкими конструкциями, чем JavaScript. QtLua не генерирует и не использует сгенерированный код для Qt. Вместо

этого она предоставляет обертки, позволяющие объектам C++ и Lua взаимодействовать друг с другом.

1. Простой “отладчик”, позволяющий просматривать глобальный индекс объекта и список загруженных сценариев.
2. Благодаря возможности перегрузки операторов, значения Lua могут быть доступны для C++
3. Из Lua могут быть доступны свойства и наследники QObject
4. Библиотека QtLua содержит функции для создания экземляров QObject и управления сигналами/слотами через Lua.

Предполагалось найти компоненту, которая строит абстрактное синтаксическое дерево и уже по нему определить тип переменных и сгенерировать код. К сожалению, такую компоненту выделить не удалось, поэтому рассмотрим другой вариант.

ToLua++

ToLua++[9] - это инструмент для интеграции кода на C/C++ с Lua. Для этого создается чистый заголовочный файл с кодом на C++ с перечислением констант, переменных, функций, классов и методов, которые мы хотим экспортировать в среду Lua. Затем ToLua++ анализирует этот файл и создает файл с исходным кодом на C/C++, который автоматически связывает C/C++-код с Lua. Если существует ссылка на созданный файл из нашего приложения, то доступ к указанному коду может быть получен через Lua. То есть в результате генерируется пакет файлов на C++, использующий API toLua и выполняющий те действия, которые были указаны. В списке литературы приведен пример[8].

Lua2c

Lua2c[4] – данная утилита преобразует код, записанный на языке Lua, в код на языке C. Данная утилита работает для большого подмножества Lua. Можно попробовать вместо выделения необходимых компонент найти транслятор из данного языка в язык C. Тогда для выполнения команд на данном языке использовать уже имеющийся интерпретатор, а при необходимости выполнения программы на роботе, производить трансляцию кода из данного языка в язык C с внесением необходимых изменений. В результате эксперимента было получено, что сгенерированный код сложен для понимания и внесения в него изменений. Ниже приведен пример:

Команда на языке Lua

```
a,b = f("how", t.x, 4)
```

транслируется в следующий набор команд на языке C:

```
lua_getglobal(L,"f");
lua_pushstring(L,"how");
lua_getglobal(L,"t");
lua_pushstring(L,"x");
lua_gettable(L,-2);
lua_remove(L,-2);
lua_pushnumber(L,4);
lua_call(L,3,2);
lua_setglobal(L,"b");
lua_setglobal(L,"a");
```

Lua_icxx

Lua_icxx[5] – это встраиваемый в приложение, написанное на языке C++, интерпретатор Lua, позволяющий легко вычислять выражения, вызывать функции, запускать скрипты на языке Lua. Позволяет легко обрабатывать несколько возвращаемых значений. Для установки требуется Lua5.1 . Пример подключения:

```
#include "lua_icxx/LuaInterpreter.h"
void main() {
        ...
        LuaInterpreter L;
        …
       LuaTableRef table = L.newTable();
       for (int i = 1; i <= 5; i++)
            table[i] = i*2;
}
```

Данный вариант нам не подходит, поскольку данная система обеспечивает только само взаимодействие, а не обработку кода.

CSL

CSL[2] (C Scripting Language) – интерпретируемый язык сценариев с синтаксисом, сходным с синтаксисом языка C. CSL – скриптовый язык с динамической типизацией. Необходимо выделить компоненту, которая строит дерево разбора, либо найти утилиту, позволяющую перевести код на языке Lua в код на языке C. Выделить компоненты не удалось, так как CSL это монолитная система и возникают аналогичные с QtLua проблемы. Трансляторов из языка CSL в C не было найдено.

Доработка существующего языка

Так как текущая реализация языка довольно-таки сырая и сложно модифицируемая, было принято решение разработать язык “с нуля”. Возможны два способа для реализации языка:

1. реализовать синтаксический анализатор методом рекурсивного спуска вручную;
2. построить анализатор по грамматике с помощью генераторов анализаторов.

В виду очевидных преимуществ был выбран второй вариант. В качестве готовых программных средств были рассмотрены:

1. ANTLR;
2. YACC;
3. QLALR.

Приведем их описание ниже.

ANTLR

ANTLR[1] - генератор синтаксических анализаторов, позволяющий автоматически создавать синтаксические анализаторы (как и лексический анализатор) на определенном языке по описанию LL(*)-грамматики на языке, близком к расширенной форме Бэкуса-Наура. По мере ознакомления с ANTLR были обнаружены некоторые неудобства. Одно из них заключается в том, что код генерящийся для рантайма платформозависим, а сам проект QReal:Robots кроссплатформенный. Другая проблема заключается в необходимости наличия пакета “autotools”, что вызывает некоторые сложности при использовании ANTLR на платформе Windows.

YACC

Программа YACC[10] (Yet Another Compiler Compiler) предназначена для построения синтаксического анализатора контекстно-свободного языка. Анализируемый язык описывается с помощью грамматики в виде, близком форме Бэкуса-Наура. Результатом работы YACC'a является программа на Си, реализующая восходящий LALR(1) анализатор. Используется совместно с генератором лексических анализаторов, например, flex или lex. В приципе YACC подходит для решения нашей задачи, но существует аналог использующий инструментарий Qt − QLALR.

QLALR

QLALR[6] − это генератор синтаксических анализаторов. QLALR использует файл грамматики и генерирует код на C++, который соответствует этой грамматике. Сам анализатор является LALR(1), т.е. с предпросмотром одной лексемы. Также, как и в YACC, при генерации выдаются ошибки о возможных конфликтах, таких как shift/reduce и reduce/reduce. Так как синтаксический анализатор, генерируемый с помощью QLALR, требует лексического анализатора, зачастую он используется совместно с генератором лексических анализаторов flex[3].

QLALR подходит нам. Рассмотрим формат входного файла с грамматикой на примере калькулятора с двумя арифметическими операциями: + и -. Имя анализатора, который будет сгененрирован QLALR, записывается в начале грамматики после директивы `%parser`.

```
%parser calc_grammar
```

После мы можем указать необходимо ли гененировать отдельно заголовочный файл и файл с реализацией. Имена выходных заголовочного файла и файла с реализацией указываются после директив `%decl` и `%impl` соответственно.

```
%decl calc_parser.h
%impl calc_parser.cpp
```

После перечисляются возможные лексемы, для этого используется конструкция `%token` <лексема>. Порядок, в котором записаны лексемы, определяет их приоритет. Данные лексемы станут терминалами сгенерированного парсера и должны быть получены из лексического анализатора.

```
%token_prefix Token_
%token number
%token lparen
%token rparen
%token plus
%token minus
```

После этого указывается стартовое состояние с помощью директивы `%start.`

```
%start Goal
```

При необходимости можно добавить код, который необходимо будет включить в раздел объявлений или в код реализации. Код, который необходимо добавить в раздел объявлений, должен быть заключен внутрь конструкции “/: :/”, а код для реализации - “/. ./”. Пример:

```
/:
```

```
        #include "qparser.h"
        #include "calc_grammar_p.h"
:/
/.
        #include "calc_parser.h"
        #include <QtDebug>
        #include <cstdlib>
./
```

Правила записываются в виде Item_1 : Item_2 (op Item_n)*. Каждая часть правила отделяется пробелом. Если существуют альтернативы, то они перечисляются по порядку, как в примере ниже. При необходимости можно вставить action-код, который будет выполнятся при выборе данного правила. Этот код заключается между маркерами “/.” и “./”.

```
PrimaryExpression : number;
PrimaryExpression: lparen Expression rparen;
/.
case $rule_number:
        sym(1) = sym(2);
        break;
./
```

Реализация

Синтаксический анализатор

Для того, чтобы построить табличный анализатор по грамматике с помощью QLALR, необходимо описать следующие компоненты.

Во-первых, описание лексем. Во-вторых, описание классов, описывающих узлы абстрактного синтаксического дерева, каждый из которых содержит ссылки на потомков и хранит информацию об операции или значение. В-третьих, записать саму грамматику в нужной форме, расставить приоритеты и ассоциативность операций.

Рассмотрим по порядку.

Лексический анализатор, обрабатывая строку, выдает в качестве результата последовательность лексем. Приведем описание всех возможных лексем:

1. plus − “+”
2. minus − “-”
3. mult − “*”
4. div −“/”
5. lparen − “(”
6. rparen − “)”
7. lsbrace − “[”
8. rsbrace − “]”
9. lbrace − “{”
10. rbrace − “}”
11. more − “>”
12. less − “<”
13. equals − “==”
14. noequals − “!=”
15. more_than − “>=”

16. less_than − “<=”
17. and − “&&”
18. or − “||”
19. bitand − “&”
20. bitor − “|”
21. bitxor − “^”
22. inc − “++”
23. dec − “--”
24. comma − “,”
25. var − имя переменной, может состоять из цифр и букв. Первый символ − буква.
26. number − число, состоит из цифр. Если число вещественное, то после будет стоять одна точка и после неё ещё одна последовательность цифр.
27. comment − состоит из символов, записанных после двух символов “/”.

Теперь рассмотрим, какие в нашем дереве могут быть узлы:

1. Node − корневой узел
2. Expression − абстрактный узел выражения
3. ExpressionConditional − узел описывающий сокращенный условный оператор
4. AssignExpression − узел для описания присваивания, который может иметь одного или двух потомков и хранит в себе знак = (или +=,-=,*=,/=)

5. Logical − узел логического выражения, хранит в себе левую и правую часть и знак логической операции
6. Condition − узел, описывающий неравенство, хранит в себе выражение условия и выражения, которые будут выполнятся в случаях then else
7. BinaryOp − узел для записи арифметических операций
8. BitWise − узел для записи побитовых операций
9. FunCall − узел для вызова функции
10. ArrayCall − узел взятия элемента из массива по индексу
11. Identifier − данный узел хранит в себе идентификатор, который может быть либо именем функции, либо переменной
12. Comment − хранит в себе комментарий
13. Number − данный узел содержит в себе число, представленное строкой
14. String − хранит в себе строковое выражение
15. Char − хранит в себе символ
16. Array − хранит в себе массив, представленный в виде списка под-деревьев разбора выражений
17. Crement − узел инкремента(декремента)
18. Arguments − узел аргументы, который хранит в себе голову и дерево разбора хвоста списков аргументов
19. Term − абстрактный узел термов.

Разберем грамматику.

```
Node : Expression;
```

Выражение может быть трех типов: либо комментарий,

```
Expression : Comment;
```

либо инкремент (декремент),

```
Expression : Crement;
```

либо присваивание значение выражения переменной.

```
Expression : AssignExpression;
```

Инеремент(декремент) состоит из идентификатора и знака инкремента (декремента) и описывается правилами, написанными ниже. Различают постфиксный и префиксный инкремент (декремент).

```
Crement : (++|-- ) Identifier;
```

```
Crement: Identifier (++|--);
```

Присваивание состоит из правой части (индетификатора), одного из знаков =,+=,-=,*=,/=, левой части − условие. И описывается правилами следующего вида:

```
AssignExpression : Identifier (=|+=|-=|*=|/=) ExpressionConditional
```

Сокращенный условный оператор состоит из выражения условия и двух выражений, которые будут выполнены в случае выполнения и невыполнения условия. Если выражение представляет из себя сокращенный условный оператор, то выполняется правило:

```
ExpressionConditional : Logical  ?  Expression : Expression;
```

иначе просто переходим к разбору выражения с побитовыми операциями

```
ExpressionConditional: BitWise;
```

Логическое условие состоит из левой и правой части и знака логической операции. Так как операторы && и || левоассоциативны, то левая часть логического выражения − неравенство, а правое − логическое условие. Разбор выражения осуществляется по следующим правилам:

```
Logical : Logical (&& | ||) Logical
Logical : Condition;
```

Неравенство состоит из правой и левой части и знака неравенства между ними. Каждая из частей представляет собой выражение с побитовыми операциями. Если знака неравенства нет, то у нас просто выражение с побитовыми операциями. Разбор осуществляется по следующим правилам:

```
Condition: BitWise (=|!=|>|>=|<|<=) BitWise;
Condition : BitWise;
```

Выражение с побитовыми операциями состоит из арифметического выражения, знака операции и выражения с побитовыми операциями. Если знака побитовой операции нет, то это арифметическое выражение. Записывается следующими правилами:

```
BitWise : BitWise (&| | ^) BitWise;
BitWise : BinOp;
```

Арифметическое выражение состоит из терма, знака операции и арифметического выражени. Если знака операции нет, то арифметическое выражение состоит из одного терма. Записывается следующими правилами:

```
BinOp : BinOp (+|-|/|*) BinOp;
BinOp : Term;
```

Терм может быть массивом, вызовом функции, числом, строкой, символом, обращение к массиву по индексу или идентификатором. Это определяется следующими правилами:

```
Term : Array;

Term : FunCall;

Term : Crement;

Term : Number;

Term : Char;

Term : Identifier;

Term : String;

Term : ArrayCall;
```

Массив состоит из списка аргументов, заключенных в фигурные скобки.

```
Array : { Arguments } ;
```

Вызов функции состоит из идентификатора и списка аргументов заключенных в скобки.

```
FunCall : Identifier ( Arguments)
```

Список аргументов состоит из выражения запятой и хвоста списка аргументов, если хвост пуст, то только из выражения. Записывается следующими правилами:

```
Arguments : ExpressionConditional , Arguments;

Arguments : ExpressionConditional;
```

Правила вывода из нетерминалов .

```
Number : number;

String :  str;

Identifier : var;
```

```
Comment : com;
```

Система вывода типов.

Заведем структуру данных, где будем хранить пару (имя переменной, тип переменной). Также после обработки инструкции у нас есть список деревьев разбора для каждой команды. Будем обрабатывать деревья по порядку. Возможны три варианта:

1. Комментарий, пропускаем
2. Инкремент (декремент) переменной, необходимо проверить, что переменная уже существует и определена, и имеет целочисленный тип. В противном случае необходимо вывести сообщение об ошибке.
3. Присваивание значения выражения переменной. Опишем алгоритм работы ниже.

Добавим переменную в список переменных, тип же определим по дереву. Каждому из листов дерева соответствует определенный тип данных.

Number − в зависимости от значения в узле может быть либо вещественным, либо целочисленным

String − переменная строкового типа

Char − переменная символьного типа

Identifier − если это не имя функции, то необходимо проверить записана ли переменная в список уже обработанных переменных. Если нет, то ошибка, иначе тип переменной берется из второго значения пары.

Array − необходимо проанализировать список аргументов. Аргументы могут быть либо строками либо символами, либо числами. Необходимо уточнить, что если аргументы являются числами и хотя бы одно из чисел вещественное, то и весь массив будет массивом вещественных чисел. В случае, если в массиве существуют элементы с различными типами, то вывести сообщение об ошибке.

Для остальных узлов тип определяется, несколькими правилами:

1. Инкрементирование (декрементирование) можно производить только над целочисленными типами.
2. В сокращенном условии типы выражений для ветки then и ветки else должны совпадать.
3. Для арифметических и побитовых операций типы правого и левого поддеревьев должны совпадать. Они могут быть либо строками, либо символами, либо числами.
4. Если оба типа числовые, но один из них целочисленный, а другой - вещественный, то результат арифметической (побитовой) операции будет иметь вещественный тип.

Генератор

Необходимо научиться генерировать код для различных роботических платформ, таких как Lego NXT (C) и ТРИК (QtScript), и в школьный алгоритмический. В каждом узле хранится знак(-и) операции, и поддеревья. Для каждого узла определяется метод, который состоит из

вывода знака(-ов) операций и вызова метода от поддеревьев в корректной последовательности. Необходимо учитывать, что в различных языках одни и те же операции могут описываются разными символами, например & в школьный алгоритмическом языке записывается как “и”, необходимо завести для каждого языка, для которого будем генерировать код, свой метод для генерации кода.

Тестирование

Был написан набор тестов для проверки синтаксического анализатора. Для начала был составлен комплект тестов для проверки корректности распознования имен переменных и чисел, как целых, так и вещественных. После этого были написаны тесты, проверяющие правильность генерации кода и построение дерева с учетом приоритетов операций.

Заключение

Было проведено исследование готовых скриптовых решений. Но, так как представленные языки в целом чересчур излишни и сложны для решения проблем, которые будут стоять перед пользователями, было принято решение разработать язык “с нуля”. Для этого было проведено исследование среди генераторов анализаторов и был выбран QLALR. После этого была переработана текстовая часть языка, разработана грамматика описывающая необходимые операции и записанная в нужной форме. По этой грамматике был сгененрирован анализатор и была проверена работа его на написанных тестах.

Результат работы https://github.com/BogdanKl/qreal/tree/TextPartLanguage

Список литературы

[1] ANTLR homepage. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files, – http://www.antlr.org/

[2] C Script Language public homesite. C Scripting Language (CSL) is a powerful and easy programming language, – sourceforge.net/http://csl.sourceforge.net/csl.html

[3] Flex is a tool for generating scanners, – http://flex.sourceforge.net/

[4] Lua2c github repository, – https://github.com/davidm/lua2c

[5] Lua_icxx docs. Lua-icxx is a C++ library that simplifies a Lua interpreter in a C++ application, – http://lua-icxx

[6]QLALR — the Qt parser generator, – http://qt-project.org/quarterly/view/qlalr_adventures

[7] QtLua project home. The QtLua library aims to make Qt4/Qt5 applications scriptable using the Lua scripting language, – http://www.nongnu.org/libqtlua/

[8]ToLua++ example, –**http://usefulgamedev.weebly.com/tolua-example.html**

[9] Tolua++ - home. Tolua++ is an extended version of tolua, a tool to integrate C/C++ code with Lua, – http://www.codenix.com/~tolua/

[10] Yacc - a parser generator, – http://luv.asn.au/overheads/lex_yacc/yacc.html